Universidade do Minho
Escola de Engenharia

Tolerância a Faltas
Maio 2023

Universidade do Minho
Escola de Engenharia

Raft + Quorum Read

Alexandre Martins (PG50168), Bruno Pereira (PG50271), Rui Braga (PG50743)

Raft + Quorum Read

Alexandre Martins (PG50168), Bruno Pereira (PG50271), Rui Braga (PG50743)

Maio 2023

1 Introdução

Os algoritmos de consenso distribuído permitem que um conjunto de máquinas funcione como um grupo coerente que consegue sobreviver a falhas de alguns dos seus membros. O *Raft* é um dos algoritmos que permitem atingir o consenso distribuído mantendo um estado coerente em todas as réplicas.

Porém, o balanceamento da carga entre as réplicas é um dos desafios do *Raft*, dado que todos os pedidos têm de passar pelo líder antes de poderem ser replicados por todas as réplicas, o que aumenta substancialmente a carga presente no líder do *Raft*.

Esta diferença de carga entre os *followers* e o líder é ainda mais óbvia quando temos uma carga de trabalho que realiza muitos pedidos de leitura, uma vez que esses pedidos serão sempre respondidos pelo líder, adicionando mais carga ao líder do *Raft*.

Portanto, uma otimização interessante para este algoritmo de consenso distribuído seria implementar leituras de quórum que permitem distribuir a carga das leituras pelos restantes nodos.

2 Implementação

Esta secção consistirá numa explicação detalhada da arquitetura e dos passos tomados para implementar o Raft, algoritmo de consenso distribuído, aliado a leituras em quórum para reduzir a carga no líder.

2.1 Raft

Nesta subsecção, começaremos por apresentar o estado necessário para a implementação deste algoritmo. De seguida, iremos apresentar os passos tomados para implementar cada uma das principais partes deste algoritmo: replicação de *logs* e eleição do líder. É importante realçar que este algoritmo corre sobre duas threads: uma focada em receber e tratar mensagens (AppendEntries RPCs, RequestVote RPCs, pedidos de leitura, etc); outra orientada a *timeouts*, nomeadamente os *timeouts* para início de uma eleição e para a execução de *AppendEntries RPCs*.

2.1.1 Representação do Estado

Para armazenar todos os dados necessários à implementação do *Raft* foi criada uma classe **State**. Nesta classe constam todas as variáveis necessárias para definir o estado de cada um dos participantes no algoritmo de consenso distribuído, sendo as variáveis mais importantes do seu estado as seguintes:

```
node_id        -> Identificação da própria réplica

node_ids       -> lista com identificadores de todas as réplicas

state          -> Representa o modo atual da réplica, uma réplica pode estar
                  num de três estados: 'Follower', 'Leader', 'Candidate'

leader_id      -> Identificação da réplica que neste momento é o líder do Raft

majority       -> Número de réplicas necessário para atingir uma maioria

stateMachine   -> Dicionário que armazena o estado de cada uma das réplicas,
                  denominado de máquina de estados no algoritmo de Raft

currentTerm    -> Mandato atual do algoritmo de Raft

votedFor       -> Réplica na qual a réplica atual votou

receivedVotes  -> Votos que a réplica atual receber

log            -> Log onde são armazenadas as operações que devem ser aplicadas
                  à máquina de estados quando o líder fizer 'commit' destas

requestsBuffer -> Lista auxiliar para armazenar os pedidos que terão de ser
                  redirecionados para o líder

commitIndex    -> Índice da entrada mais recente cuja replicação no log já
                  foi confirmada por uma maioria de réplicas

lastApplied    -> Índice da última entrada que foi aplicada à máquina de estados
                  na réplica atual

nextIndex      -> Dicionário em que cada chave corresponde a um identificador
                  de uma réplica e o valor corresponde ao índice da próxima
                  entrada a ser enviada para essa réplica

matchIndex     -> Dicionário em que cada chave corresponde a um identificador
                  de uma réplica e o valor corresponde ao índice da última
                  entrada que se sabe que foi replicada no log dessa réplica

randomizer     -> Variável utilizada para gerar números aleatórios

lock           -> Lock utilizado para controlo de concorrência entre as threads
```

2.1.2 Replicação de Entradas do Log

No algoritmo de consenso distribuído ***Raft*** as operações propostas pelos clientes são armazenadas num **log** no nodo líder, que tem a responsabilidade de as replicar por todas as réplicas do sistema.

O processo de replicação das entradas do **log** começa sempre com a receção de um novo pedido para alterar a ***stateMachine***. Quando este pedido é recebido por uma réplica que não se encontra no estado ***Leader***, esta tem de redirecionar este pedido para o ***Leader***, caso exista, caso contrário deve armazenar o pedido no **requestsBuffer** - os pedidos armazenados neste ***buffer*** serão redirecionados para o ***Leader*** assim que um seja elegido.

O líder do ***Raft*** ao receber uma nova proposta para alterar o estado, vai adicionar essa proposta ao seu ***log*** sob a forma de uma ***log_entry*** que é um par *(message, currentTerm)*.

O líder, para informar as restantes réplicas do sistema que continua ativo, envia mensagens periódicas a todas elas - estas mensagens são denominadas de ***heartbeats***. Estas mensagens servem não só para manter as réplicas informadas sobre o estado do líder (se continua ativo, ou se se assume que falhou), mas também replicar o estado do líder nas réplicas do sistema.

Os ***heartbeats*** correspondem na verdade a um ***AppendEntriesRPC*** que não carrega nenhuma entrada do ***log*** do líder para ser replicada, sendo que uma mensagem **AppendEntriesRPC** encapsula os seguintes parâmetros:

```
term         -> corresponde ao mandato atual do líder
entries      -> novas entradas para serem replicadas
leaderID     -> Identificador da réplica que corresponde ao líder do Raft
prevLogIndex -> corresponde ao índice da entrada que precede as novas entradas
                presentes em 'entries'
prevLogTerm  -> mandato correspondente à entrada no índice prevLogIndex
leaderCommit -> índice da entrada do log até à qual se sabe estar replicado
                em todas as réplicas
```

Tendo em conta que as mensagens ***AppendEntriesRPC*** contam também como *heartbeats*, o líder, após ser eleito começa de imediato a enviar periodicamente estas mensagens para todas as réplicas do sistema, e sempre que existam entradas novas no seu ***log*** para replicar, estas serão encapsuladas nas mensagens para todas as réplicas. Uma réplica no estado ***Follower*** ao receber uma mensagem de ***AppendEntriesRPC*** vai começar por verificar se o mandato recebido na mensagem é superior, ou igual ao mandato atual da réplica. De seguida, e de forma a obedecer à propriedade do algoritmo *Raft*, **Log Matching**[1], verifica-se, utilizando o ***prevLogIndex*** e ***prevLogTerm*** contidos no *RPC*, se o *log* da réplica está igual ao do líder até a esse índice.

Após isso, é necessário adicionar as novas entradas enviadas pelo líder ao ***log*** da réplica, para isso, é necessário verificar se já existe alguma entrada no ***log*** com o mesmo índice, se sim vamos verificar se o mandato está de acordo com o mandato do líder. No caso de estar de acordo, remove-se essa entrada e todas as que a seguem do ***log***.

Finalmente, após adicionar as novas entradas recebidas do líder (***entries***) e se atualizar o ***commitIndex*** da própria réplica, é necessário agora responder ao líder, e esta resposta será encapsulada numa mensagem do tipo ***appendEntriesRPCResponse*** que engloba os seguintes parâmetros:

```
term             -> Representa o mandato atual da réplica.
success          -> Indica se a operação teve ou não sucesso.
                    (assume apenas dois valores Verdadeiro ou Falso)
nextIndex        -> Próximo índice do log que o 'Follower' deseja receber.
buffered_messages -> Lista de mensagens que foram recebidas dos clientes,
                    que serão agora redirecionadas para o líder.
```

Caso alguma das verificações acima referidas falhe, o parâmetro ***success*** será enviado a ***False*** e o líder saberá que algo correu mal, sendo que existem dois cenários possíveis para esta operação não ser completada com sucesso: a última entrada conhecida como replicada na réplica, pelo líder não existe; ou o *term* da réplica é superior ao *term* do líder.

Começando pelo caso em que a entrada que o líder pensava estar replicada na réplica não existe, então o líder terá de decrementar o ***nextIndex*** correspondente a esta réplica e reenviar uma mensagem do tipo

[1] (*Log Matching* é a propriedade do algoritmo Raft que garante que se dois *logs* contêm uma entrada cujo índice e o mandato são iguais, então os logs são iguais até a esse índice.)

AppendEntriesRPC para esta réplica, e este processo vai repetir-se até que o líder e a réplica estejam de acordo quanto à última entrada replicada, mesmo que para isso seja necessário reenviar todas as entradas do ***log*** do líder novamente. Quanto ao segundo cenário, quando o mandato da réplica que está no estado ***Follower*** é superior ao mandato do líder, então esta réplica ao responde à mensagem do líder, envia-lhe, para além da resposta negativa, o seu próprio *term*. Ao analisar a resposta, o líder vai verificar que se encontra num mandato desatualizado, atualiza-o para o mesmo valor recebido, converte-se para *Follower* e aguarda a receção de um **heartbeat** vindo do líder. Ao fim de um certo período, se não receber qualquer **hearbeat**, iniciará um processo de eleição como é explicado em detalhe na secção 2.1.3.

2.1.3 Eleição do Líder

No algoritmo de consenso distribuído **Raft**, após um certo intervalo de tempo sem receber qualquer mensagem (”AppendEntries RPC”) do líder, uma réplica, que de momento se encontra no modo *Follower* ou *Candidate*, transita para o modo *Candidate* no caso de se encontrar no modo *Follower*, incrementa o mandato (variável ”term”), vota para em si próprio e envia uma requisição de voto para todas as outras réplicas (**RequestVote RPC**). Os **RequestVote RPCs** são acompanhados do mandato e índice da última entrada presente no *log*, para que o próximo líder seja a réplica com o *log* mais atualizado.

Uma réplica, aquando da receção de um **RequestVote RPC**, começa por verificar o quão atualizado o nodo que enviou o *RPC* está, através da verificação do mandato (”currentTerm”) contido nela. Existem três situações possíveis:

1. O mandado da réplica é **inferior** ao mandato presente no *RPC*: Nesta situação, independentemente do modo em que a réplica se encontre, começa por converter-se para *Follower*, atualizar o seu mandato para o mesmo mandato da mensagem. De seguida, o nodo passa a comparar o índice e mandato da última entrada do seu *log*, com o índice e mandato contidos no *RPC*, para determinar quem contém o *log* mais atualizado[2]. Se a réplica que enviou o *RPC* contém o *log* mais atualizado, então o nodo vota na réplica que lhe enviou o *RPC*. Para votar, o nodo guarda numa variável (”votedFor”) a identificação do nodo no qual votou neste novo mandato (para poder rejeitar outro pedido de voto, para um mesmo mandato, vindo de outra réplica), e envia uma resposta para o votado informando que votou nele (define o valor do campo ”voteGranted” da resposta como *True*).

2. O mandado da réplica é **igual** ao mandato presente no *RPC*: Este caso pode ser divido em três subcasos, um por cada modo em que a réplica se encontre:

   (a) **Follower**: Assim como acontece no primeiro caso, o nodo verifica quem é que tem o *log* mais atualizado antes de responder que votou ou não votou na réplica candidata;

   (b) **Candidate**: Responde negativamente, indicando que não votou no nodo que enviou o *RPC*;

   (c) **Leader**: Não responde ao RPC, envia um *AppendEntries RPC*, para informar o nodo perguntador, da sua existência como líder do Raft.

3. O mandado da réplica é **superior** ao mandato presente no *RPC*: Responde negativamente, envia o seu mandato atual[3], para que o nodo perguntador possa atualizar-se.

Nesta secção já falamos dos procedimentos de envio e receção dos **RequestVote RPCs**. Para acabar, apenas falta referir como são tratadas as respostas a estes *RPCs*. Quando uma réplica recebe uma resposta ao *RequestVote RPC* enviado, começa por verificar o quão atual a resposta é. Se a mensagem for de um mandato anterior ou se a resposta for negativa, i.e., se o nodo que respondeu não garantiu o voto, então a mensagem é ignorada. Se o mandato da resposta for igual ao mandato atual, se o nodo ainda se encontrar no modo *Candidate*, e se a resposta for positiva, i.e., se o nodo que respondeu garantiu o voto, então o nodo regista o novo voto e reinicia o *timeout* de início de uma nova eleição. Quando forem recolhidos votos da

[2] O *log* mais atualizado é aquele cuja última entrada possui o maior mandato, ou no caso de os mandatos serem iguais, ganha quem tiver o maior *log*.

[3] Todos os RPCs relativos ao Raft e as suas respostas contêm o campo ”currentTerm”, para se poder verificar o mandato do nodo que enviou a mensagem.

maioria (incluindo o próprio nodo), então o nodo transita para o modo **Leader**, envia **heartbeats**[4] para todas as réplicas, e adiciona ao *log* as entradas recebidas de clientes durante o processo de eleição.

2.2 Leituras de Quórum

As leituras são operações que no algoritmo do ***Raft*** podem influenciar o desempenho de todo o sistema em casos que a carga no sistema seja muito elevada, isto deve-se ao facto de que este algoritmo de consenso distribuído apresenta um líder explícito, e todas os pedidos de leitura e escrita têm de passar por este líder.

Desta forma, o cliente ou contacta sempre o líder, ou no caso de contactar outra réplica que não seja o líder, os pedidos são redirecionados para o líder, contribuindo para uma grande concentração de carga do sistema apenas no líder, impulsionando o desbalanceamento de carga em todo o sistema.

Os pedidos de leitura em específico representam ainda uma maior influência para este desbalanceamento de carga, uma vez que no algoritmo de ***Raft*** estes pedidos só são respondidos pelo líder, então um aumento drástico de peidos de leitura ao líder poderia levar a um aumento de latência na resposta a outros pedidos e a todo o processo de replicação das réplicas que é sempre iniciado pelo líder.

Portanto, uma solução para este problema seria descentralizar os pedidos de leitura utilizando para isso leituras de quórum, e assim as leituras poderiam ser feitas a qualquer um dos nodos do sistema. Para este novo tipo de leituras, uma réplica ao receber um pedido de leitura poderia proceder de uma das seguintes formas:

- Caso o modo da réplica seja ***Leader***, este pode responder imediatamente ao pedido de leitura
- Caso contrário, se a réplica se encontrar em modo ***Follower*** ou ***Candidate***, este poderá ou reencaminhar o pedido para o líder, ou executar uma leitura de quórum (única opção quando o se encontra no modo ***Candidate***).

Analisando agora o segundo cenário, em que a réplica contactada pelo cliente apresenta o estado de ***Follower*** é necessário escolher entre reencaminhar o pedido para o líder, ou realizar uma leitura de quórum. Isto é feito através de probabilidades, ou seja, existe uma variável ***probReadFromLeader*** (calculada em função do número de réplicas) que representa a probabilidade da réplica ler diretamente do líder. Para além disso, quando a réplica decide executar uma leitura de quórum é escolhido um conjunto de nodos aleatório (que não inclui o líder) aos quais serão enviados os pedidos de leitura, sendo que o número de réplicas escolhidas tem de ser suficiente para que contando com a própria réplica, se consiga atingir a maioria. Não incluir todas as réplicas no quórum de leitura, é também um mecanismo que tem como objetivo diminuir a carga nas diversas réplicas.

[4] Heartbeats são **AppendEntries RPCs** no qual não são enviadas entradas do *log*. Servem para informas as restantes réplicas de que o líder está ativo, e dessa forma impedir novas eleições.

3 Testes e Conclusão

Nesta secção vamos descrever os vários testes realizados para demonstrar não só que as funcionalidades do algoritmo de ***Raft*** funcionam e foram implementadas de forma correta, mas também uma comparação de *performance* entre a versão normal do algoritmo e a versão com leituras de quórum.

No que toca aos testes de comparação de *performance*, serão realizados vários testes, começando por cargas menores e progressivamente aumentando a carga no sistema para tentar visualizar as diferenças no que toca ao débito do sistema e à latência das respostas a cada um dos pedidos.

3.1 Leitura Não Linearizável

No algoritmo de **Raft**, quando um cliente deseja realizar uma leitura, efetua o pedido a uma das réplicas, se essa réplica for o líder então este responde diretamente, caso contrário o pedido é reencaminhado para o líder, deste modo pode-se afirmar que os pedidos de leitura são sempre respondidos diretamente pelo líder. Dado que, o líder é aquele que armazena a informação mais atualizada em cada momento do sistema, ao realizar um pedido de leitura a um líder, vamos receber a informação mais atualizada. Contudo, quando se trata da versão do algoritmo com leituras de quórum, um pedido de leitura realizado por um cliente pode ser respondido por um conjunto de réplicas que não incluí o líder. Por isso, é possível que ao responder ao pedido de leitura do cliente, utilizando um quórum de leitura, a operação, apesar de já ter sido confirmada por todas as réplicas e aplicada no líder, pode ainda não ter sido aplicada nas réplicas pertencentes ao quórum, e dessa forma provocar uma resposta com um valor desatualizado, o que não seria linearizável. Na figura 1, presente nos anexos, é possível observar um exemplo de uma leitura não linearizável.

3.2 Mudança de Líder

No que toca à eleição do líder, é possível observar nas imagens 2 e 3, presentes na secção 4.1 todo o processo de eleição, desde que um elemento do sistema inicia uma eleição, depois de exceder o seu *election timeout* até ao momento em que um novo líder é eleito. As figuras estão também acompanhadas de uma descrição mais detalhada de todo o processo.

3.3 Comparação entre Raft sem e com leitura de quórum

De modo a perceber quais os impactos de *performance*, foram realizados vários testes com diferentes taxas de pedidos por segundo de modo a variar a carga colocada sobre as réplicas do sistema, com o objetivo de provocar diferenças no desempenho das duas versões, sendo que o esperado era que com o aumento de stresse colocado sobre o sistema, as leituras de quórum conseguissem distribuir a carga pelas réplicas do sistema de forma mais equilibrada, libertando o líder ***Raft*** e aumentando a sua capacidade de responder a pedidos. Os testes realizados, variando o número de nodos, concorrência e número de pedidos por segundo, foram os seguintes:

1. número de nodos: 5, número de pedidos por segundo: 500
2. número de nodos: 5, número de pedidos por segundo: 1500
3. número de nodos: 10, número de pedidos por segundo: 500
4. número de nodos: 10, número de pedidos por segundo: 1000

3.3.1 Teste 1

Para este teste, assim como foi mencionado na secção 3.3, para este teste o *maelstrom* foi configurado com os seguintes parâmetros:

```
--node-count  -> 5 nodos a perticipar no Raft
--concurrency -> 2n (o número de clientes corresponde a duas
                 vezes o número de nodos)
```

```
--rate        -> 500 pedidos por segundo (aproximadamente).
--time-limit  -> 300 segundos
```

Observando os gráficos de latência, para o primeiro teste, representados pelas figuras 4 e 5 consegue-se verificar que as latências dos pedidos de leitura para a versão com e sem quóruns de leitura são muito semelhantes, verificando-se que a versão com quóruns de leitura apresenta latências ligeiramente superiores, mas trata-se de uma diferença pouco significante - esta diferença no que toca às latências entre as duas versões pode estar relacionada com o facto de que quando se realiza uma leitura de quórum, existem mais mensagens envolvidas do que numa leitura feita diretamente ao líder, o que pode atrasar um pouco a resposta.

3.3.2 Teste 2

Para este teste, assim como foi mencionado na secção 3.3, para este teste o *maelstrom* foi configurado com os seguintes parâmetros:

```
--node-count  -> 5 nodos a perticipar no Raft
--concurrency -> 2n (o número de clientes corresponde a duas
                 vezes o número de nodos)
--rate        -> 1000 pedidos por segundo (aproximadamente).
--time-limit  -> 300 segundos
```

Aqui, tal como aconteceu no primeiro teste os graficos de latência para os vários pedidos, representados pelas figuras 6 e 7, apresentam grandes semelhanças no que toca às latências dos pedidos de leitura, sendo que entre o teste 1 e o teste 2, duplicou-se a taxa de pedidos realizada por segundo.

3.3.3 Teste 3

Para o terceiro teste realizado, decidiu-se aumentar também o número de nodos envolvidos no algoritmo de raft de modo a verifica se isso teria algum impacto na performance do sistema. Deste modo, o teste foi realizado com os seguintes parametros no maelstrom:

```
--node-count  -> 10 nodos a participar no Raft
--concurrency -> 2n (o número de clientes corresponde a duas
                 vezes o número de nodos)
--rate        -> 500 pedidos por segundo (aproximadamente).
--time-limit  -> 300 segundos.
```

Foi então duplicado o número de nodos a participar no algoritmo de ***Raft***, sendo os resultados apresentados nas figuras 8 e 9 nos anexos, e para este número de nodos obteram-se novamente resultados muito semelhantes no que toca às latências dos pedidos de leitura, sendo a versão sem leitura quóruns consegue latências ligeiramente melhores, mas a diferença entre as duas versões não é muito notória.

3.3.4 Teste 4

No quarto e último teste, foi utilizado o mesmo número de nodos do teste anterior, contudo duplicou-se a taxa de peiddos por segundo, na tentativa de provocar alguma diferença notória entre as duas versões do algoritmo de ***Raft***. Sendo assim, os parâmetros utilizados no maelstrom foram os seguintes:

```
--node-count  -> 10 nodos a participar no Raft
--concurrency -> 2n (o número de clientes corresponde a duas
                 vezes o número de nodos)
--rate       -> 1000 pedidos por segundo (aproximadamente)
--time-limit -> 300 segundos
```

Os resultados deste último teste podem ser observados nas figuras 10 e 11. Analisando estes gráficos, é possível observar que a versão do algoritmo sem leituras de quórum apresenta latências menores para os pedidos de leitura, para este caso a diferença, apesar de não ser muito elevada, é mais notória do que nos últimos três testes.

3.4 Conclusões

Face a todos os testes apresentados é possível afirmar que o algoritmo de ***Raft*** implementado cumpre com todos os requisitos necessários para o seu correto funcionamento e ainda com a implementação de leituras de quórum, tal como pedido no enunciado prático.

No que toca à comparação entre as versões sem e com leituras de quórum, as diferenças entre as duas versões não foram tão significativas quanto o esperado, no que toca à latência dos pedidos de leitura, no entanto a implementação da leitura de quórum continua a ser um ponto importante para distribuir de forma mais eficiente a carga pelas diferentes réplicas participantes no sistema, que contribuirá para um melhor desempenho do sistema.

4 Anexos

4.1 Leitura não linearizável

Na figura abaixo podemos verificar o momento em que o líder do Raft recebe da maioria, a confirmação de que a operação ”write key: 0, value: 4” foi replicada no log. Nesse momento, o líder incrementa o seu **commitIndex**, aplica a operação na sua máquina de estados e informa o cliente de que a operação de escrita foi realizada com sucesso. Logo após esta confirmação, podemos verificar que uma leitura em quórum é realizada, e resulta no envio, para o cliente, de um valor diferente daquele que consta no líder. Esta leitura de um valor passado aconteceu no intervalo entre os AppendEntries RPCs do líder, o que se traduz na não receção do valor atualizado do **commitIndex** do líder que resultaria na aplicação da última operação de escrita confirmada pelo líder. Esta leitura é designada de não linearizável já que num processo sequencial após a confirmação da escrita, não seria possível ler um valor diferente daquele que foi confirmado na escrita.

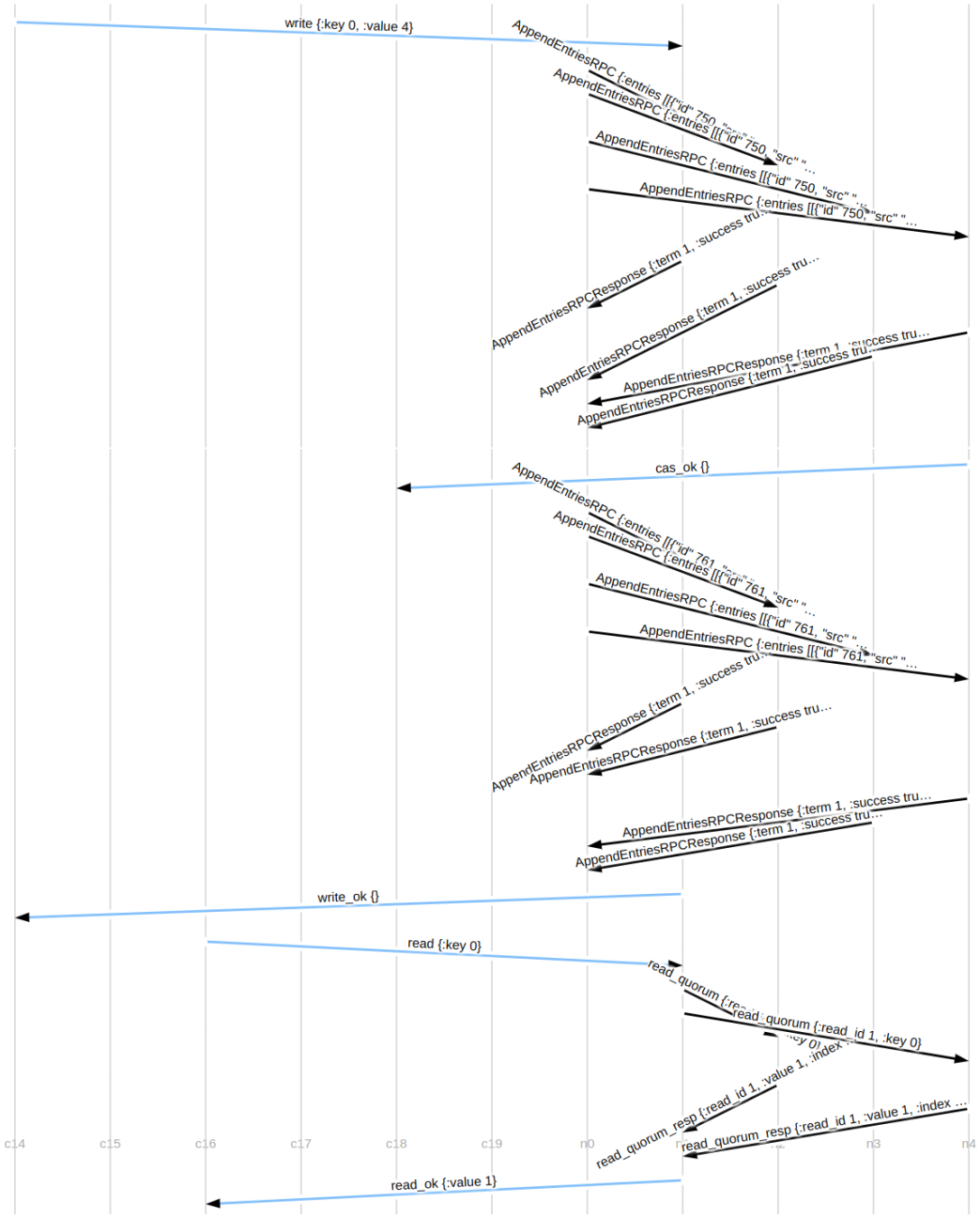

Figura 1: Exemplo de leitura não linearizável

4.2 Mudança de líder

O exemplo que se segue é composto por duas imagens. Neste exemplo é possível observar um líder que perdeu o seu cargo, mas que logo de seguida o recuperou.

Para melhor perceção dos acontecimentos, o teste foi realizado com apenas 3 servidores e 6 clientes. Enumerando a partir da direita, na imagem, os primeiros 3 nodos são servidores designados por: ”n2”, ”n1”, ”n0”.

No início da primeira imagem, podemos verificar que o ”n2” não recebe qualquer mensagem, isto acontece devido à existência de uma partição que isola o nodo ”n2”, deixando-o sozinho. Desde o início do teste que ”n2” se encontra sozinho, tendo o seu mandato aumentado progressivamente com os sucessivos *timeouts* de eleição. Como podemos ver, quando a partição desaparece, podemos verificar que ”n2” recebe um AppendEntries RPC que imediatamente rejeita devido ao seu mandato ser substancialmente superior. O líder ao receber a resposta ao *RPC*, percebe que o seu mandato está desatualizado e passa imediatamente para o modo ’Follower’. Logo após esta conversão, o nodo ”n2” vai executar uma sucessão de tentativas de eleição, tentativas que não vão ter sucesso, dado que o seu *log* se encontra vazio, e portanto, os outros servidores não o aceitam como líder. Finalmente, na parte final da segunda imagem, podemos verificar que o nodo ”n1” inicia uma eleição. Eleição que vai ser suportada por todos os outros servidores, já que este possui o *log* mais atualizado. Podemos verificar que a eleição foi um sucesso, já que ”n1” começa, novamente, a enviar *AppendEntries RPCs*.

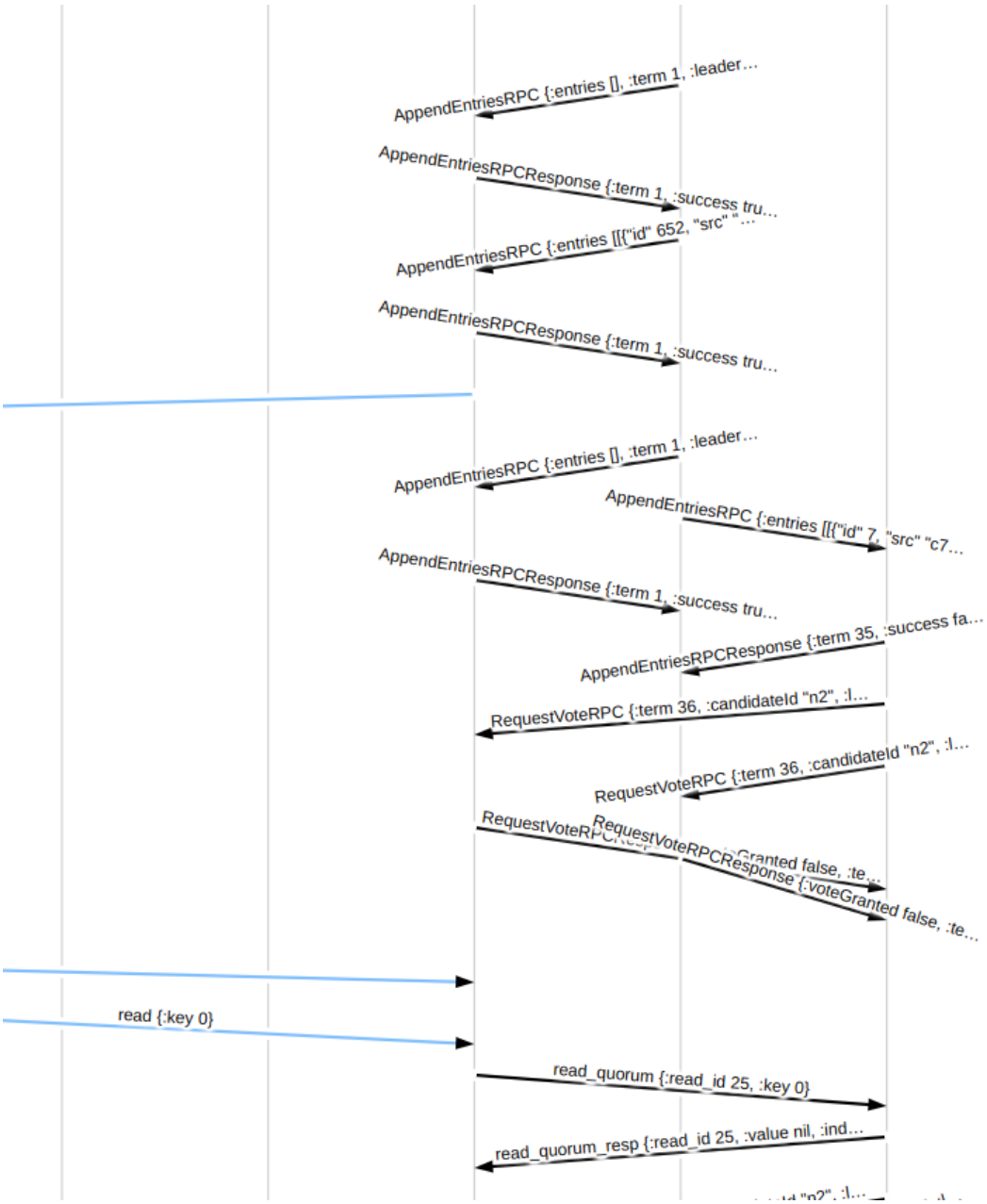

Figura 2: Eleição de líder: 1ª metade

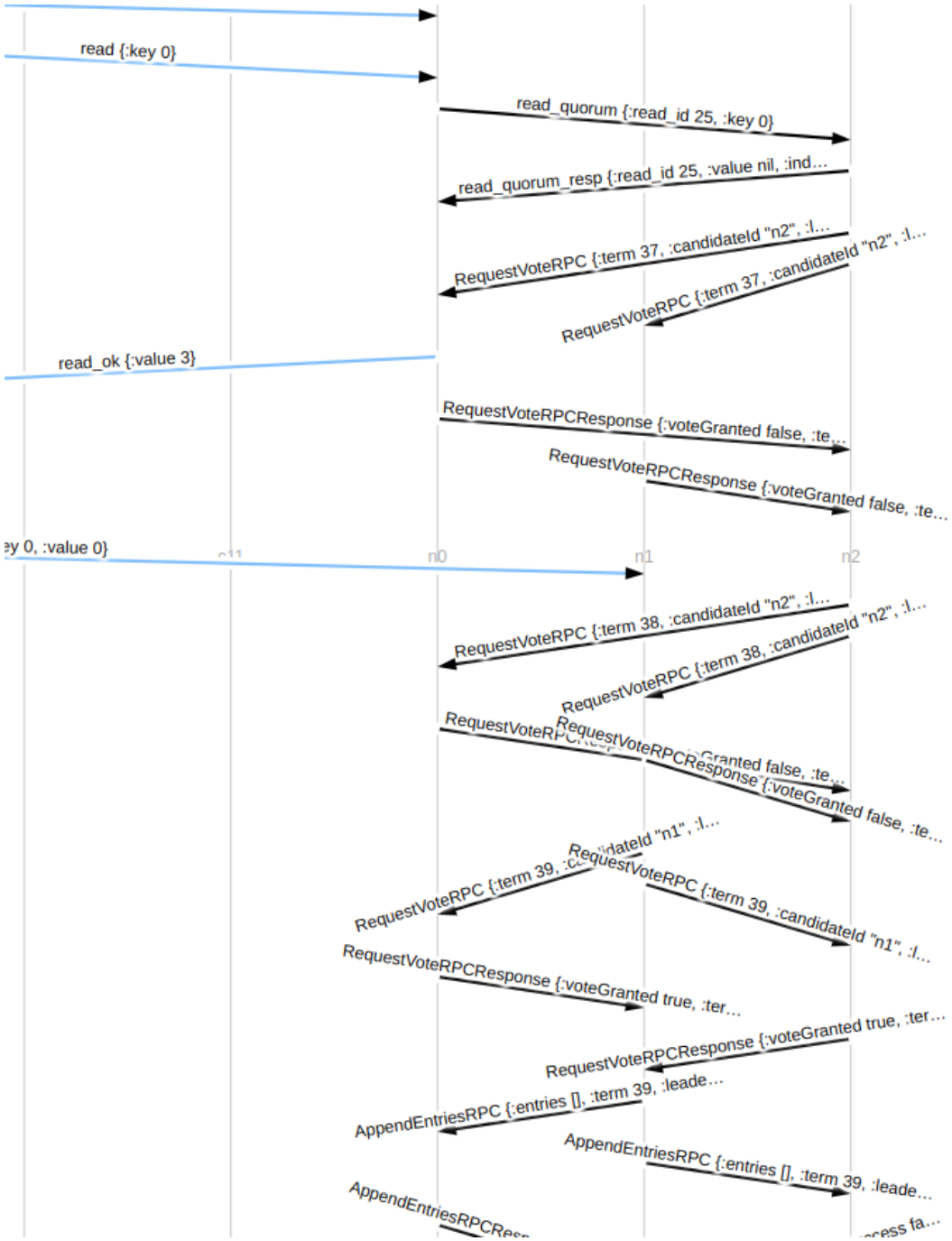

Figura 3: Eleição de líder: 2ª metade

4.3 Comparação de desempenho entre Raft sem e com leitura de quórum

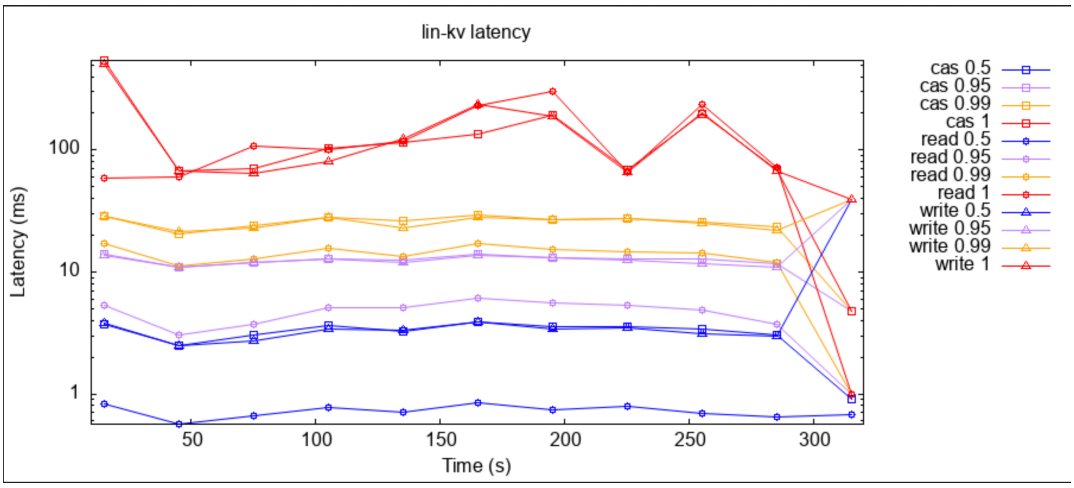

Figura 4: Gráfico ”latency-quatiles” obtido no maelstrom, para o primeiro teste, com leituras de quórum.

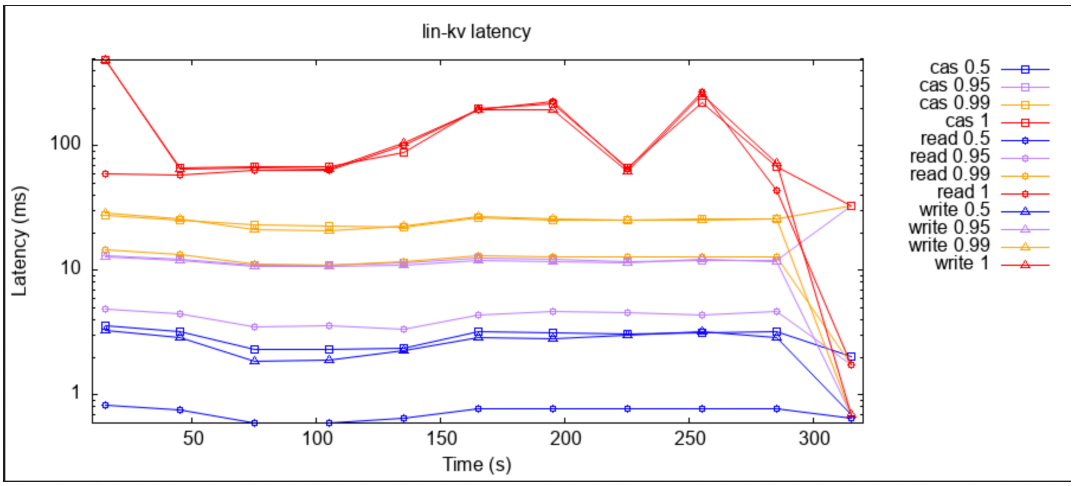

Figura 5: Gráfico ”latency-quatiles” obtido no maelstrom, para o primeiro teste, sem leituras de quórum.

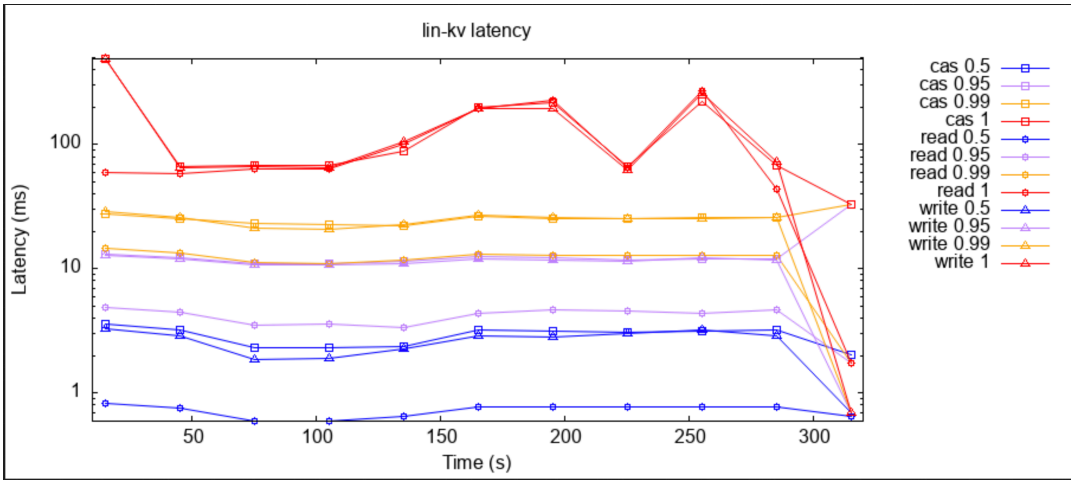

Figura 6: Gráfico ”latency-quatiles” obtido no maelstrom, para o segundo teste, sem leituras de quórum.

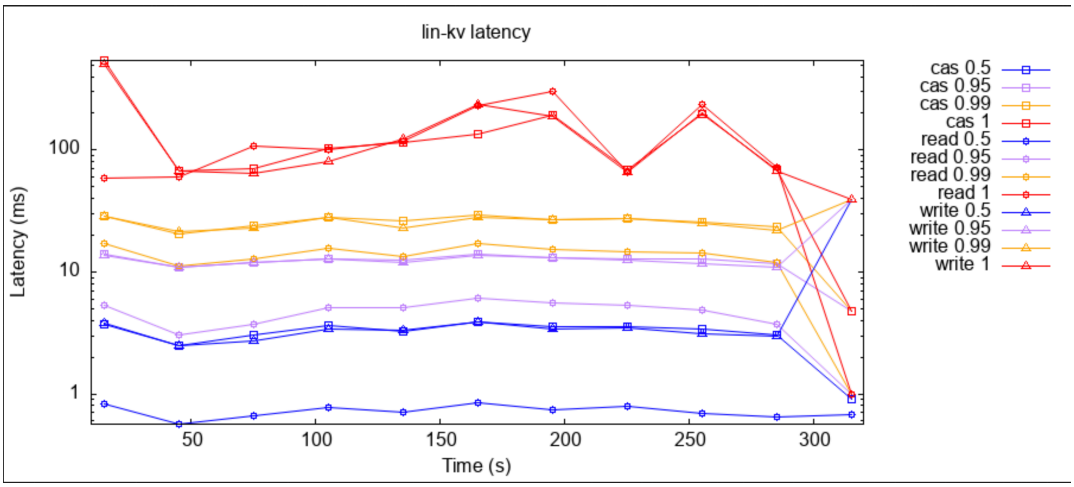

Figura 7: Gráfico ”latency-quatiles” obtido no maelstrom, para o segundo teste, com leituras de quórum.

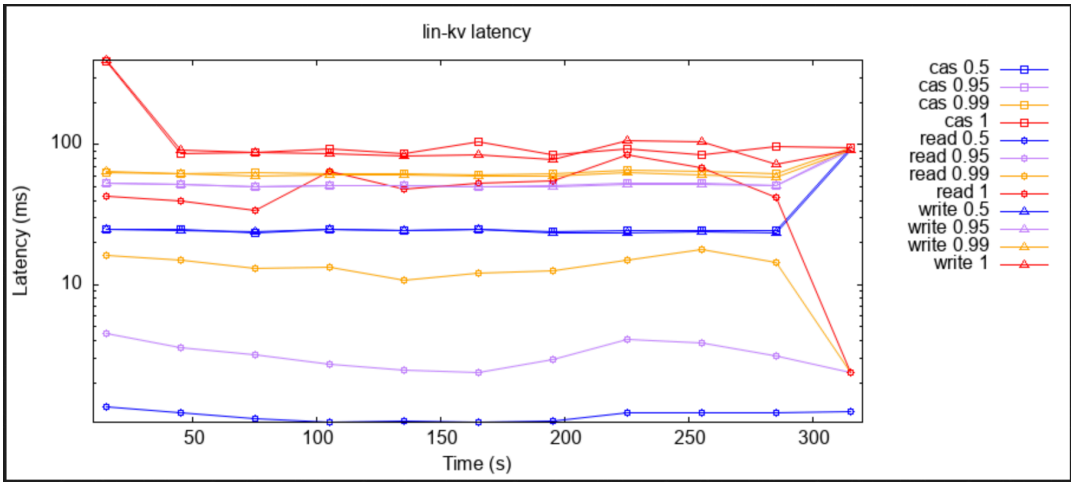

Figura 8: Gráfico ”latency-quatiles” obtido no maelstrom, para o terceiro teste, sem leituras de quórum.

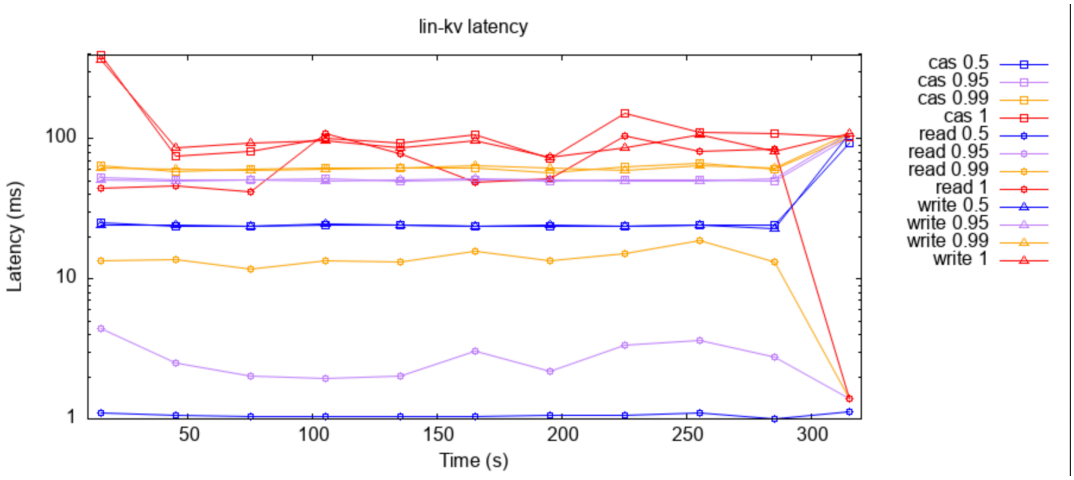

Figura 9: Gráfico ”latency-quatiles” obtido no maelstrom, para o terceiro teste, com leituras de quórum.

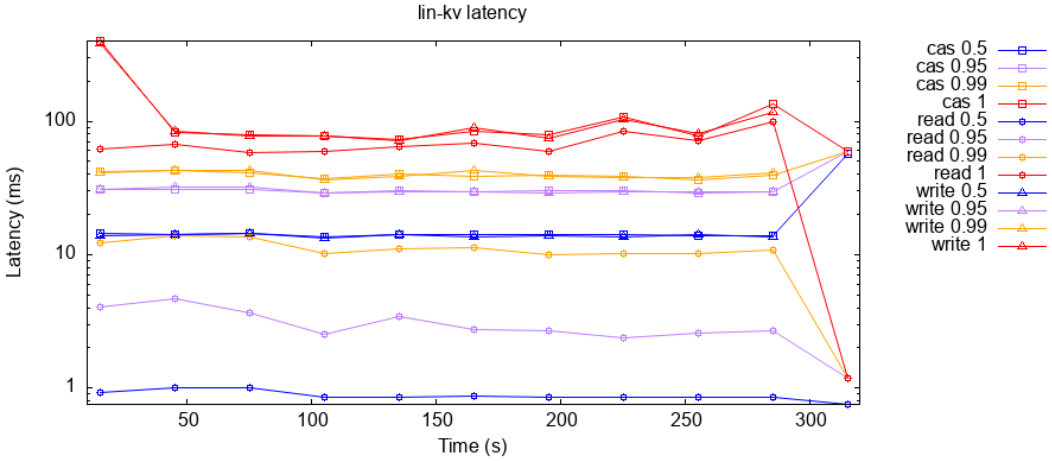

Figura 10: Gráfico ”latency-quatiles” obtido no maelstrom, para o quarto teste, sem leituras de quórum.

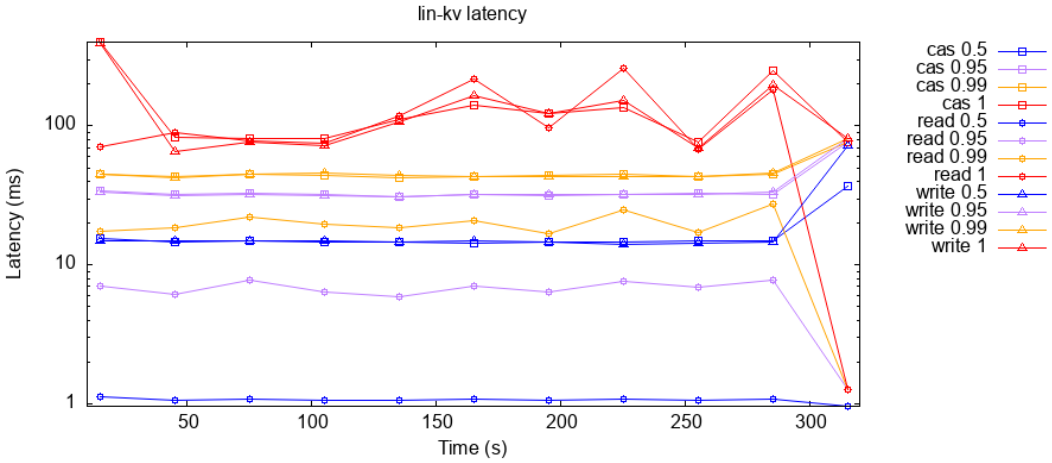

Figura 11: Gráfico ”latency-quatiles” obtido no maelstrom, para o quarto teste, com leituras de quórum.

5 Notas

No código fonte que acompanha este relatório, podemos verificar a existência de duas versões:

- **raft.py**: Versão que implementa o **Raft** e as leituras em quórum;
- **raft_only.py**: Versão que implementa o **Raft** e que consulta o líder em todas as leituras.